can cause your application to behave inconsistently with other software on the system.

Visual Focus

Many accessibility aids need to identify the "focus point" where the user is working. For example, a blind-access utility must describe the text or object that the user is working on, and a screen-magnification utility pans out to enlarge whatever is at a particular point on the screen. Other utilities may move pop-up windows, so they do not cover "where the action is."

Sometimes, it is easy for an accessibility aid to determine this location; the operating system provides it when it moves the focus between windows, menus, or standard controls. It is more difficult to determine the location when an application uses its own method of indicating the visual focus *within its window*, such as highlighting a cell in a spreadsheet, an icon, or a windowless custom control. In these cases, to be accessible, the application must make its focus location available to other programs in the system; the convention for doing this is to move the system caret.

Moving the System Caret

The system caret is the blinking vertical bar that the user sees when editing text, but it can be placed anywhere on the screen, made any shape or size, and even made invisible. If it is invisible, it can be moved to indicate the focus location to applications without disturbing what the user sees on the screen.

Making the system caret invisible is easy: simply call the **CreateCaret** function to set the caret's size and shape and the **SetCaretPos** function to move it to wherever you are drawing the visual focus indicator (the highlighted cell, icon, button, and so on). Note that it is present but invisible, unless you explicitly make it visible.

An application should only display focus and selection indicators when they are in the active window. When the window loses activation, the application should remove the visual indicator and also call the

DestroyCaret function to inform other applications. (For Win32 applications, this step is not strictly necessary, but is still good practice.)

Determining the Keyboard Focus

Sometimes, it is hard to decide what to indicate as the focus location. Extended and discontiguous selection often confuse the issue, but the keyboard focus location should be considered independent of selection, even if an application normally links the two.

The following examples may clarify the distinction and help you learn to identify and indicate the keyboard focus location in your own application:

- **Insertion bars in text.** When the user moves an insertion point within text, it is usually drawn with the real system caret. If the application chooses to draw its own insertion point, it should still move the system caret invisibly, tracking the location of the visible insertion bar.
- **Extended text selection.** When the user makes an extended selection, one end of the selection is always the "active" or moving end, and that is the actual location of the keyboard focus. For example, to select three characters, you start with the insertion bar in an edit control, and then hold down the SHIFT key while pressing the right arrow twice. The end where you started is the "anchor," the stationary end; at the right, you should see the flashing caret marking the active end. If you hold down the SHIFT key and press another arrow key, it is the active end that moves, and that is where the system caret should be placed. You should display a visible insertion bar at the active end, because that is useful feedback for all users.
- **On graphic objects.** When a user moves the keyboard focus to a graphic object, such as an icon or a bitmap, an application should place the system caret invisibly over the same object so that the caret's rectangle covers the entire image. If there is an adjacent label, the caret should cover that as well.
- **Within graphic objects.** Sometimes, an application uses a single bitmap to represent several objects, such as a group of graphical buttons. The application usually indicates the focus by highlighting a portion of the bitmap, drawing a dotted rectangle over it, or even

moving the mouse pointer. In addition to indicating the focus, the application should also place the system caret invisibly over the region of the bitmap that corresponds with the "hot spot" or object being referenced.

- **Simple controls.** If an application is drawing simple custom controls, such as a custom push button, the keyboard focus is associated with the entire control, so the entire control should be covered by the system caret. (This is necessary only for windowless custom controls. If the control is a window, the window takes the keyboard focus, so it is not necessary to identify it using the system caret.)
- **Complex controls.** A complex or composite control, such as a list box, can place the focus on individual elements within the larger control. In this case, an application should use the system caret to indicate the area of the particular item that has the focus. Even though the application might think of the collection of items as a single control, they should be treated as separate control elements when they are identified to external components.
- **Spreadsheets.** When the user navigates within a spreadsheet, the focus is usually placed on an entire cell, rather than on content within the cell. Often, this is indicated by a bold cell border, and the application should place the system caret over the entire cell. If the user begins editing the contents of a cell, the application should indicate the focus appropriately for the content text or graphics.
- **Discontiguous selection.** Discontiguous selection is usually supported among discrete items, rather than in text. There is always one item that has the keyboard focus or was most recently clicked by the mouse, and that object should be covered by the system caret. To see an example, select an item in a folder or in File Manager, and then hold down the CTRL key while pressing arrow keys to move the focus rectangle to a file that is not part of the selection.
- **Mouse-only objects.** Although applications should provide keyboard access to all their functionality, some objects can only be manipulated or selected using the mouse. In this case, you should treat the object when it is selected as if it received the keyboard focus and use the system caret appropriately to indicate that it has the focus. Of course, if the real keyboard focus moves, you should

follow it, because the mouse-only object is no longer the object of the user's attention.

Controls and Menus

A sighted user can usually identify a control, such as a push button or check box, by its appearance, but a user who is blind has to rely on a screen review utility to describe the object in words. The utility presents the user with the name, type, and state of the control. For example, after the user has tabbed to a check box, the utility might say "Quick Printing check box, checked." Obviously, it can only do this if it can determine all the properties from the application. Voice input utilities and on-screen keyboards have similar requirements; they need to identify and name specific controls and to determine how to manipulate the control in response to the user's commands. In some cases, the use of nonstandard controls can render an application completely unusable for users who rely on accessibility aids.

Controls can be divided into these categories, each of which will be discussed in the following sections:

- Standard Windows controls (including Windows common controls).
- Owner-drawn controls, which behave like standard controls, but have a customized appearance.
- Superclassed standard controls, which add customized behavior to a standard Windows control.
- Custom controls, which are implemented by an application without using the normal Windows mechanisms.
- OLE Controls, which are custom controls designed to a standard programming interface.
- Owner-drawn menus.

Standard Windows Controls

Standard Windows controls should be used whenever possible, because they are the most compatible with accessibility aids.

Each standard Windows control is a separate window of a specific class, so the accessibility aid can get notified when the focus moves to a new

control. The aid can determine the control's window class, and that tells the aid what additional messages it can send to the control to query or alter the state. The aid can also identify all of the child controls contained within a parent window and identify the parent of any control.

The Windows 95 common controls library provides standardized implementations of many controls that are not supported by Windows itself, and these are all designed to be compatible with accessibility aids. Many, such as the list view and tree view controls, are extremely flexible and can be used to replace a variety of custom and owner-drawn controls.

Owner-Drawn Controls

Owner-drawn controls can be accessible as long as care is taken in their use. Although owner-drawn controls behave like standard controls, they have a customized appearance. Some applications use custom controls to change the appearance of a control, but owner-drawn controls are also an acceptable, and more accessible, option. For example, an application using an owner-drawn control might display a check box with an actual check mark instead of an X or label a push button with a picture instead of a word. Using a standard Windows control with the owner-drawn style makes the control appear normal to accessibility aids, but still allows the application to give control elements a customized appearance.

You should define the label for an owner-drawn control, even if the label text will not be visible on the screen. If you create an owner-drawn control in which the normal caption will not be visible (for example, a button with a graphic face) and leave the caption as a blank string, the accessibility aid will not be able to query the caption with a WM_GETTEXT message and use it to identify the control. You should make sure that your owner-drawn control handles all the other class-specific text retrieval messages, such as CB_GETKBTEXT, LB_GETTEXT, and so on, and set the appropriate style bits (such as LBS_HASSTRINGS) to indicate that the owner-drawn control supports those messages.

Superclassed Standard Controls

Some applications that use standard controls alter their behavior by employing a technique known as superclassing. When an application uses superclassed standard controls, basic control functions are still handled by the underlying system code for the standard control type, but the application adds its own special behavior. You should follow these guidelines when using superclassed standard controls:

- Make sure that the superclassed controls respond to the normal messages for their class.
- Use recognizable class names. Because superclassed controls normally have a unique class name, you should make sure the control's class name identifies the base class by including the normal class name as part of its name. For example, a superclassed button could be given a class name like "MyAppButton." Any accessibility aid encountering this name would assume that the control is a superclassed button.
- Do not include the name of an unrelated standard class in a superclassed control's class name because an accessibility aid might mistakenly assume that the control is related to the standard class.

Custom Controls

You should generally avoid using nonstandard custom controls, because they are not fully usable with screen review or voice recognition utilities. Custom controls present a number of problems, because accessibility aids cannot identify the type of the control or its state. In addition, if the control does not have its own window, accessibility aids are not able to watch it receiving and losing focus.

At this time, there is no standard way for applications using nonstandard controls to work well with accessibility aids. However, you can use these techniques as short-term solutions:

- If the custom control has its own window, you can return a descriptive string when the control is queried using the WM_GETTEXT message. For example, a control that appears as a button with the label Print could return the string "Print button" to convey both the control type and the label. The same string would be appropriate if the control looked like a button, but had a graphic showing a printer rather than a textual label. Likewise, a custom

control that behaves like a check box could return a caption string "Printing Enabled check box, checked."

- If the custom control has no window, you can associate a descriptive string with the control by using the techniques described in "Use of Bitmapped Text" later in this document. This string can follow the same conventions described in the previous paragraph.
- If the custom control has no window, you can convey the focus location to accessibility aids by moving the system caret as described previously in "Visual Focus."
- When using custom controls that have their own window, you should support the WM_GETDLGCODE message, which identifies the keyboard input that is supported and also the equivalent standard control if there is one. The one form of custom control that is regarded as accessible is OLE controls. For more information, see the following section.

OLE Controls

The preferred method for creating custom controls is to use the OLE controls architecture. OLE controls are an extension of the model used by the Microsoft Visual Basic programming system. Each control is an OLE object with many standardized properties and methods. Future OLE interfaces will support a method of obtaining interface pointers (object handles) to each object in a window or region of a window by querying with a window message. Once you have an interface pointer, you can make calls to an object to retrieve or set properties, including the object's location, name, and primary value.

An application that hosts OLE controls should support the **IOLEContainer** interface, which allows the enumeration of embedded controls and other objects.

Owner-Drawn Menus

You should always provide an alternative to owner-drawn menus, especially if they have a purely graphical appearance.

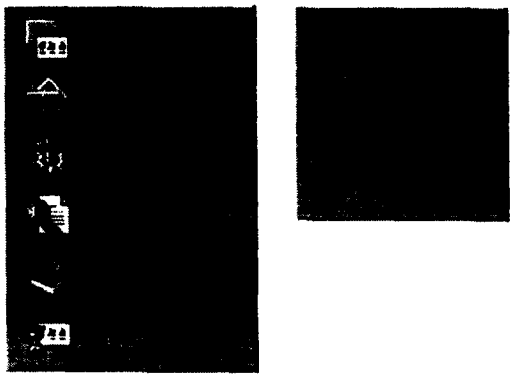
Owner-drawn menus are a flexible means of presenting a customized appearance, by making a menu item a combination of graphics and text,

or graphics alone. For example, a menu might allow the user to select from a series of colored rectangles or select a line thickness from examples, which are easier to understand than purely textual representations (such as 3 points, 4 points, and so on).

However, owner-drawn menus are incompatible with most types of accessibility aids that need to identify the names of each menu item. Therefore, you should provide an option that replaces graphics menus with standard, textual menus when an accessibility aid is being used. The following techniques can be used as short-term solutions:

- Menu items can be redesigned to include both graphics and text. For example, a menu item for selecting line width might display a sample of a line followed by text stating the width. This redundancy would also be useful for a sighted person doing layout based on a written standard, which requires lines to have a particular thickness.
- The user can be given a choice of graphics or text. If he or she chooses text, the application can revert to standard rather than owner-drawn menus. This is a good use for the Screen Reader Present flag, a global flag that tells all applications when the user is relying on a screen review utility. When that flag is set, the application can use the textual values instead of (or in addition to) the graphical presentation.

The following illustration contrasts the use of an owner-drawn menu using text and graphics with the use of a standard menu using only text.



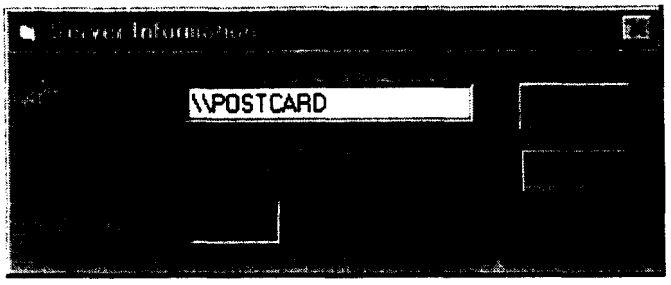
Using Appropriate Controls for Displaying Information

Windows provides the following controls for displaying information to the user: static controls, read-write edit controls, read-only edit controls, and status controls

Making the proper choice of control can improve the keyboard access to your application as well as its compatibility with accessibility aids. To choose a control that is appropriate for the user's interaction with the information, follow these guidelines:

- Use static controls for labels. In the illustration that follows, the words "Server:," "Comment:," and "Current Users:" identify accompanying controls and are displayed as static controls.
- Use read-write edit controls for values that the user can edit directly. In the illustration that follows, the server name is an edit control that starts with a default value, which the user can freely change. Read-write edit controls should always have visible borders.
- Use read-only edit controls for values that the user cannot edit. In the illustration that follows, the comment about the server is represented by a read-only edit control, which allows the user to select the text and copy it to the clipboard or to drag it to another document. This control is also included in the tab order so that a user can navigate to it using the keyboard. (Note that the label for read-only edit controls is not required to have underlined access keys.)
- Use status controls from the common controls library to display values that may change dynamically as the user watches. In the illustration that follows, the number of current users can change. In addition to being consistent with other applications, these controls provide information to screen review utilities, which may notify the user when values change.

The following illustration shows the static, read-only edit, write-only edit, and status controls described in the preceding list.



Drawing Operations

The type of drawing operation used can affect compatibility with screen review utilities. Screen review utilities watch calls to drawing functions and remember what text and graphics have been drawn and where. They also check textual attributes, such as font, size, formatting, and so on. In addition, they watch information being copied from one location to another and being erased or overwritten by other text or graphics. These utilities rely on being able to monitor normal Windows drawing operations.

Drawing Using the Standard Windows Functions

An application should always draw text using the standard Windows function calls, such as **ExtTextOut**, so the drawing can be seen by screen review utilities. This is true whether drawing is to a screen or to an off-screen bitmap. By watching every function call that creates a bitmap, draws to it, or copies from it, text can be tracked until it is displayed to the user.

Two techniques, the use of bitmapped text and the direct manipulation of pixels or bitmap bits, bypass the normal system calls and prevent a screen review utility from working. These techniques are discussed in the sections that follow.

Use of bitmapped text

Some applications ship precreated bitmaps either as resources or in separate files shipped with the application. A screen review utility can watch these bitmapped images being loaded, manipulated, and copied to the screen, but it has no way of deciphering the contents of the bitmap.

If the bitmap contains text, the text will be visible to the sighted user. The utility, however, will not be able to present the text to the user who is blind. These bitmaps are called "bitmapped text." Because the text starts out as part of the bitmap, it is never available to the system.

An application should draw text using the standard Windows functions. There may still be cases, however, where it is necessary to display text as a hand-tuned bitmap (an example would be a corporate logo). There are two simple methods that the application can use to inform the screen review utility about the text associated with the bitmap. The first method is to use a tooltip control to associate the label with the area of the screen where the bitmap is drawn. The second method is to draw the text over the bitmap. The application can easily carry the text along with the bitmap, most likely as entry in its string table resource. When the bitmap is loaded from disk by using the **LoadBitmap** function, the application can also load the text by using the **LoadText** function. It can then inform the screen review utility of the relationship between the two.

To perform a drawing operation, create a temporary screen-compatible bitmap and an associated memory device context (DC). Draw the text into the bitmap by using the **ExtTextOut** function, and then use the **BitBlt** or **StretchBlt** function to copy the bitmap onto the destination location, specifying the NOP raster operation. The destination can be either the screen or another off-screen bitmap. It is easiest to do this operation when the bitmap is first loaded into memory. The accessibility aid tracks the information subsequently.

It is only necessary to perform extra operations when a screen review utility is running. To determine whether one is running, call the **SystemParametersInfo** function with the SPI_GETSCREENREADER value.

Direct manipulation of pixels or bitmap bits

An application should use standard Windows functions, such as **BitBlt**, **PatBlt**, and **StretchBlt**, for erasing text and other graphic objects and for erasing or copying bitmap memory. If an application must use alternative means, you should provide an option for reverting to standard behavior.

Some applications directly manipulate the memory associated with a DC, bypassing the Windows functions altogether. This is most commonly done with monochrome or device-independent bitmaps. When Windows functions are not used, however, the screen review utility is not aware of the changes taking place. For example, if an application draws text into a bitmap using a Windows function call and then later erases it by clearing the bitmap memory, the screen review utility will assume that the text is still present. If the bitmap is used again for another operation, the text might be read to the user, even though it is no longer visible. Similarly, if the bits comprising one bitmap are copied directly into another without using the Windows functions, the screen review utility will not be aware of it, and text displayed visually might be unseen by the screen review utility.

The Windows application programming interface (API) provides several means of manipulating bitmap or display pixels directly, such as DirectDraw, Display Control Interface (DCI), WinG, and the **CreateDIBSection** function. These techniques bypass screen review utilities. If your application relies on these techniques for performance, you also support using more conventional methods when a screen review utility is running on the system. To determine whether one is running, call the **SystemParametersInfo** function with the SPI_GETSCREENREADER value.

Identifying Separate Screen Areas

Anything that is drawn using a single operation appears to a screen review utility as a single object. For that reason, a bitmap appears as one object, even though it might look like several distinct objects to the sighted user. An example is a custom control that looks like an array of buttons, but is really a single bitmapped image. A screen review utility describes the entire array as a single object, so the user has no way to manipulate the individual buttons.

In such cases, the application should use a tooltip control to identify each separate region. The tooltip control is one of the Windows common controls introduced in Windows 95. It identifies a region by displaying the textual label associated with it. It provides this

information to the sighted user in a way that is consistent with the rest of the Windows interface.

If, for some reason, you cannot use tooltip controls, you can use the following two techniques to identify regions of the screen. However, both are less functional and less standardized than the tooltip approach:

- An application can draw each component object as a separate bitmap. Normally, this will not impact performance, memory, or disk space. Objects can be drawn to an off-screen bitmap and then copied to the screen in a single operation.
- An application can keep an array as a single bitmap, but identify the separate regions for a screen review utility using a tooltip control.
- An application can keep an array as a single bitmap, but identify the separate regions for a screen review utility by drawing shapes over the bitmap invisibly using a NOP raster operation. This technique is described in "Use of Bitmapped Text."

It is only necessary to perform these extra operations when screen review software is running. To determine whether one is running, call the **SystemParametersInfo** function with the SPI_GETSCREENREADER value.

Identification of Windows

Windows need to be identified for the user and for accessibility aids so that the function of windows can be determined.

Identifying Windows for the User

You should try to assign a user-friendly caption to every window, whether or not it is visible on the screen.

Every window can have a caption, whether or not it has a visible caption bar. Screen review utilities query this text by sending a WM_GETTEXT message and use it to identify the window to a user who is blind when the window receives focus or when the user issues a "What window am I working in?" query. Similarly, voice recognition and on-screen keyboard utilities use the caption as a command that the user can choose. However, for this to work, the application developer

has to provide appropriate text when the window is created or by calling the **SetWindowText** function.

Note that the need for captions applies to all windows—not only top-level windows but also to child windows, such as floating palettes, custom controls, and toolbars—and to panes within the same window frame when they are implemented as separate windows.

Identifying Windows for Accessibility Aids

You should try to give different types of windows separate, unique window classes so that accessibility aids can identify their function.

Accessibility aids sometimes need to specialize their handling of different windows within the same application. For example, an application that has both a word-processing window and a spreadsheet window might draw the visual focus indicator in very different ways. Screen review utilities may also have separate instructions for handling these windows, such as identifying areas of the window that should automatically be read to the user whenever they change. Voice input and on-screen utilities also allow the user to choose between names to move focus to the window.

While a human being can identify a window by its title, this is not a reliable mechanism for accessibility aids, because many window titles change dynamically with the document or the status, or are localized into different languages. The window class name does not change under either circumstance.

Timing and Triggering Events

In some cases, people with disabilities may have difficulty accessing information because of the time and duration that information is displayed. In general, you should allow users to customize timings and avoid triggering events that could cause unexpected results. The following sections describe basic problems related to timing and triggering of events.

Adjusting General User Interface Timings

Any timed behavior should be adjustable by the user. Some individuals have slower than average reaction times, and it can be difficult for them to use features that rely on fixed timings. Examples of this include the autoscrolling that takes place when the user drags towards the edge of a window or holds down the mouse button over a scroll bar. In some cases, you should allow the user to turn off timed behavior altogether. An example of this would be any event that happens automatically when the mouse or keyboard focus pauses over on an object for specific amount of time.

Message Time-outs

You should avoid having messages time out. However, if you must use time-outs, you should provide an option to disable them in your application.

There are many reasons why a user may not spot a warning that is only displayed for a brief period of time. For example, the user might be using a screen enlarger and may have to reposition the viewport or adjust other attributes to read the text correctly. He or she may take a longer time than normal to type in an answer or take a longer than average time to read and understand the message. The user might even step away from the desk for a moment.

If a message is really important, the best way to make sure it is seen it to display it until the user consciously dismisses it. Even if a message is unimportant, it is disconcerting to have the message disappear before it can be read. If the user does not have time to fully read a message, how can he or she determine whether it is unimportant?

Flashing

You should flash objects and text on the screen only at the caret blink rate. Flashing at certain rates can cause epileptic seizures, but people susceptible to such seizures can adjust the caret blink rate through Control Panel to a rate that is harmless for them. By flashing objects and text only at the rate that the user has specified, an application can prevent causing such seizures.

Triggering of Events by Mouse Pointer Location

You should avoid having events triggered by movements of the mouse pointer over or off a special area. If you must include triggering, you should make it optional.

Some accessibility aids require the mouse pointer to move when information is explored on the screen. For example, a screen review utility may move the mouse to follow words being read, or a user may need to move the mouse to enlarge certain text.

Reliance on a mouse pointer can be a problem if movement of the mouse causes unexpected results. For example, if text appears when the mouse moves over an object and disappears when the mouse moves off of it, the text essentially disappears each time the user tries to read it!

There are two cases where it is acceptable to trigger changes based on mouse pointer movement, because these cases are already understood by accessibility aids and handled appropriately:

- It is acceptable to change the shape of the mouse pointer as it is moved. For example, you can change the shape to indicate whether or not an object is a valid drop target.
- It is acceptable, and in fact encouraged, to use tooltip controls to display an object's name or other explanatory information when the pointer is paused over an object. However, this is only the case if a standard tooltip control is used rather than a custom implementation.

Triggering of Events by Keyboard Focus Location

You should avoid having events triggered by movements of the keyboard focus. However, if you must include triggering, you should make it optional.

To enable the user to "read" or explore a window's contents, you should support keyboard mechanisms that allow the focus to change to a control or area without causing problems. For example, it is typical for a user who is blind to use the TAB key to move through all the controls in a dialog box as a means of exploring it before he or she goes back and does any actual work.

There are exceptions to this rule, primarily in cases where application behavior has been standardized and mechanisms exist for accessibility aids to work appropriately:

- It is acceptable to display explanatory text that gives details about the function of a menu while menu messages are being processed. It is preferable to draw this text in a status bar to be consistent with other applications, but any text drawn during menu processing will be assumed to serve this function.
- It is acceptable to automatically change the value of option controls (radio buttons) and tab controls during keyboard navigation. Although this behavior can cause problems for keyboard users, it is necessary for backwards compatibility.
- An application that takes some action when the focus moves should provide an alternative way to move the focus. This is typically done by using the CTRL key to modify the navigation key. For example, in Windows Explorer or in a list box that supports discontiguous selection, the user can move the focus and change the selection when navigating with an arrow key. However, the user can move the focus without changing the selection by holding down the CTRL key when he or she presses the arrow key.

Color

The use of color can greatly enhance a user interface, but only if it is used appropriately. In general, your application should not convey information by color alone, because some people will not be able to see it. Color should only be used to enhance, emphasize, or reiterate information shown by other means. If your application must convey information through color, you should make sure it is also available through some another means.

Customizing Color

For many people, color is a matter of preference. They may use Control Panel to choose a personal color scheme that they enjoy, but they do not mind and probably do not even notice if an application always draws its elements in a fixed color.

However, for many users with visual impairments, color is critical. Many people require a reasonably high contrast between text and the background to be able to read. They may even need a particular scheme, such as white text on a black background, to prevent the background from "bleeding" over and obscuring the foreground text. Some people consider the default color scheme quite legible, but find that it causes eyestrain over longer periods of time. Still others, nearly 10% of males and 1% of females, have some form of color blindness that makes certain color combinations unreadable.

Conveying Information by Color Alone

If your application conveys information by color alone, some users will not be able to make use of the information. Even allowing the user to customize the colors is insufficient if the user can only read white text on a black background or if the user is using a hand-held computer with a monochrome display. For these situations, the application should also make the information available through a means other than color.

Using Standard System Colors Where Appropriate

When possible, an application should use the standard system colors that the user has selected through Control Panel. This is easiest to accomplish when an element in the application's window corresponds to a usage handled by Control Panel, such as window text, button face, dialog box text, and so on. By using the color combinations that the user has explicitly chosen, you reduce the chance that your choice of colors will make your application unusable, and you ensure that your application's colors are pleasing to the user without having to provide a user interface for adjusting colors. For a complete list of system colors, see the description of the **GetSysColors** function in the Microsoft Win32 Software Development Kit (SDK).

Your use of the color and the use selected in Control Panel do not need to correspond exactly. For example, the user's choice of window text color and background is probably a safe combination to use for any purpose.

Using Colors in Proper Combination

Your application should always use system colors in their proper foreground-background combinations to ensure that they have reasonable contrast. The user will never choose a button text color that is the same as the button face color, so these will always be legible when used together. However, the user may alter the color scheme so that system colors that normally contrast, such as button text and window background, might be the same color on their systems. If your application draws using colors that are not specifically designed to be used in combination, the information may be completely invisible.

Your application should always draw foreground objects in foreground colors and fill backgrounds with background colors. Many users require specific high-contrast combinations, such as white text on a black background, and drawing these reversed, as black text on a white background, causes the background to "bleed" over the foreground. This combination can make reading difficult, or even painful, for some users.

The following list shows some combinations that are safe to use and others that are not.

Status	Type of combination
Safe combinations	Window text on window background
	Button text on button face
Unsafe mixing combinations	Window text on button face
	Button text on window background
Unsafe reversed foreground and background	Window background on window text
	Button face on button text

Making Custom Colors Customizable

If you use colors for elements that do not correspond to system colors selected in Control Panel, you should provide your own means for adjusting the colors. For example, you could design a calendar application that uses different background colors to indicate various types of events. When using application-specific colors in this way, you

should allow the user to assign his or her own choice of colors for the elements.

Historically, some applications have had fixed colors to prevent the user from selecting an "ugly" color scheme that would make the application look unattractive. However, a user will not complain about a color scheme that he or she chooses, but may be displeased by a fixed color scheme.

Another option is to provide patterns as an alternative to colors. In the case of the calendar application, users could be allowed to choose a pattern along with the color for each type of scheduled event. They could choose a color combination that works for their eyes and supply any additional information as a background pattern. This option works best when the pattern fills an object without interfering with the text.

Coloring Graphic Objects

Graphical objects present special challenges. For example, some application display buttons that have pictures on them instead of, or in addition to, text. Do the colors selected in Control Panel apply to this case?

If the picture on the button is monochrome, the answer is simple. The button face should always be drawn in the standard system color (the COLOR_BTNFACE or COLOR_3DFACE button value), and the foreground image should be drawn in the standard button text color (the COLOR_BTNTEXT button value). If the image is drawn inside a window rather than on a button, it is more appropriate to use the COLOR_WINDOW and COLOR_WINDOWTEXT values instead of the button colors.

A multicolored picture presents more problems. The easiest solution is to include a monochrome image that can be used on monochrome displays or that can be used when the user has chosen a nondefault button face color or has requested High Contrast Mode (described later in this document).

If you cannot include monochrome images, you can try creating them on the fly from the multicolored images by identifying light and dark areas as foreground and background. For example, a bitmap that has a

multicolored object on a white background could be mapped with all colors other than white in the appropriate system foreground color and with white in the system background color. These colors could be reversed for images designed with a dark background.

Preventing Backgrounds from Obscuring Text

Text drawn over a varied background, such as a wash of colors or a bitmap, may be illegible for some viewers, so you should always provide the user with the option to omit the image and revert to a plain background. Text is most legible when drawn against a plain background of a contrasting color, and many users with low vision will not be able to read text if the background is irregular.

Instead of providing an option to control contrast in your application, you can simply omit the background in response to the High Contrast Mode setting, which is discussed in the following section. You should also omit the background if the foreground color changes. For example, text drawn over a very light bitmap image might appear quite legible in the default color scheme, but be unreadable if the user chooses a light foreground text color.

You can keep a complex background reasonably legible by making sure that the background image contrasts well with the text. If the foreground text is black, many users will find it hard to read if it is drawn over areas of the image that are brown or other dark colors.

High Contrast Mode

An application that uses standard system colors or allows the user to choose colors that are not defined by the system has its basic color-related needs covered. However, Windows 95 introduces a new feature called High Contrast Mode, which the user can activate through Control Panel to advise applications to provide high contrast visuals. Applications can check for this setting by calling the **SystemParametersInfo** function with the SPI_GETHIGHCONTRAST value. Applications should query this value during initialization and when processing WM_COLORCHANGE messages.

When the High Contrast Mode flag is set, an application can take additional steps to make its display friendly for users who require high contrast. You should use these techniques when the High Contrast Mode flag is set:

- Omit bitmapped images or other complex backgrounds behind text and controls.
- Draw images in monochrome instead of multiple colors, and draw them using standard foreground and background colors.
- Replace application-specific colors with standard system colors defined through Control Panel, and try to use as few color combinations as possible.

Size

Many people like to fit a maximum amount of information onto a single screen. However, there are also many who find small type difficult, or even painful, to read. Users with severe visual impairments will probably choose accessibility aids, such as a screen enlarger, that can zoom in on a portion of the screen. A much larger number of users may have no trouble reading 12 point text on the computer screen, but find it difficult to read text in smaller sizes. Other users may not have trouble reading text, but suffer from headaches and eyestrain at the end of a day. These users do not view small type as a problem of accessibility; they view it as one of usability.

Although an application might work fine for a user on one computer, it may be unusable on another computer. Applications that are designed to look good at standard resolution may have their information shrink to near invisibility when run on a high resolution monitor, or information may appear inaccessibly off the screen when run on a small, pen-based computer.

Selectable Font Sizes

The best way to satisfy users who prefer small type and those who require larger type is to allow them to choose the typeface and size that best fit their needs. This simple feature can make applications seem more user-friendly.

The preferred approach is to provide a menu option or property sheet where the user can choose the font using the standard Font Selection dialog box. A second approach is to automatically resize the fonts when the user resizes the window, but this approach is less flexible because the user cannot use a large font in a small window with scroll bars.

The following example illustrates selectable font sizes. The FaxFire application is used to drive a fax card connected to a user's computer. It has a window that shows a list of all the faxes that the user has sent or received during the last month with each line showing information for a single fax: its date, destination, and so on. The line has a maximum length of 50 characters, and because it is always drawn in a 10 point font, the window is a fixed size. However, some users have complained about the fixed font size. How can the font size be fixed?

With a minimal amount of work, a Font command can be added to one of the menus that enables the user to choose a font using the standard Font Selection dialog box provided in the Windows common dialog box library. When the application draws its text, it uses the font that the user has requested. In case the font selection makes some of the information extend beyond the edge of the window, the window can be changed to be resizable and be given scroll bars. The total amount of work required to fix the font sizing problem is quite reasonable.

Providing Alternatives to WYSIWYG

Some applications try to present a WYSIWYG ("what you see is what you get") view of a document, making the text on the screen reflect the appearance that the text will have on the printed page. However, some people may want to print text in a tiny font, but not edit it when it is that small. In reality, the size of type on screen need not be linked to the size that the text will be when printed. It is easy to allow the user to adjust the size of information on the screen through several methods, such as draft and zoom modes, which are described in the following sections.

Draft mode

One method for allowing the user to bypass WYSIWYG is to provide a draft mode, which uses a single font for all information. This mode customarily uses a single type of annotation, such as underlining, to

indicate characters that would normally be drawn with any form of additional formatting, such as bold or italics. (Draft mode also provides an added benefit for users running on extremely slow systems or with little free memory, because it typically performs better in those situations.)

Ideally, you should allow the user to choose the draft font. Using the system font may be the best way to conserve memory, but it might not be the best for the user's vision.

Zoom features

An extremely valuable feature that applications can support is a "zoom" facility, which scales everything in the document to a user-selected ratio. Many applications, such as Microsoft Word and Microsoft® Excel, offer this feature, and it is beneficial to many users who do not consider themselves to have disabilities as well as those who do. Use of the TrueType® scalable font technology ensures that characters will remain clearly defined at almost any size.

Scaling Nondocument Regions

Most applications display information in more forms than just text. Buttons, rulers, and graphic images can also pose problems for people if the object is a fixed size (especially a small size).

Many application windows contain two types of information: an image of a document created by the user and one or more panes belonging to the application itself. A good example of nondocument panes is a word processor's toolbar of command buttons. If the Zoom command applies only to the document and not to the surrounding information, the user may still find it difficult to use the application or suffer from "tiny button syndrome." The application could have a single zoom factor applied to both types of information. However, the size of buttons should not change when 8 point text is zoomed because much of the benefit of a toolbar is lost if the user has to scroll to reach some of the buttons.

A better solution is to allow the user to independently select the zoom ratio for each pane, whether a document or nondocument region. For

example, "Toolbar Size" can be provided as a separate option, and a single setting can apply to all toolbars. It is also acceptable to provide a simple option that permits the user to choose from a range of sizes instead of using a more general scaling factor.

Compatibility with System Screen Metrics

It is important that applications drawing their own screen elements pick up the size settings that the user has selected in Control Panel. For example, a private dialog box manager that draws custom dialog boxes should use the dialog box font that the user has selected for the rest of the system. The same principle applies for scroll bars, custom menus, and so on. For a complete list of size settings, see the description of the **GetSystemMetrics** function in the Win32 SDK.

Line Width

Although many applications draw lines with a fixed width of one pixel, those lines disappear on high-resolution monitors. They may also be invisible to a person with low vision. Instead of using fixed widths, applications should determine the proper thickness of a line by calling the **GetSystemMetrics** function with the SM_CXBORDER and SM_CYBORDER values. These values are defined appropriately for the resolution of the monitor, and in future operating systems the user will also be able to adjust them appropriately for their vision.

Global Scaling

Windows 95 provides a Custom Font Size feature—that is, the ability to globally scale all fonts and most other visual elements on the screen by changing the number of pixels used to represent a "logical inch." To be compatible with this feature, applications should avoid drawing in MM_TEXT mode, which bypasses logical scaling. If an element of the application's user interface uses MM_TEXT while the rest does not, that one element will be drawn out of proportion to all the rest of the screen elements.

It is important to note that bitmaps are not automatically scaled by this factor. Bitmaps are discussed further in the next section.